Local Applications Table

Support for closed loops, etc.

Aug 28, 1990

A means of expansion of the VME local station system software is by the use of local applications, which are separately-compiled procedures that are invoked by the system during Data Access Table processing. The Local Applications Table (`LATBL`) contains the entries that result in their invocation.

Invocation context

During `Update` task processing of the Read Data Access Table (`RDATA`), one particular entry causes the `LATBL` entries to be processed. This means that local applications are invoked during updating of the datapool, likely to be positioned either at or near the end of `RDATA`.

Each local application (LA) is expected to be written as a Pascal procedure, just as are application page programs. The calling sequence is as follows:

```
TYPE
TrigType = (init, term, kbint, cycle, net);
ParamList = RECORD
              sVarPtr: sVarPtrType;
              enableBit: Integer;
              params: ARRAY[1..9] OF Integer;
             END;
PROCEDURE LocalApp(trig: TrigType; VAR LAEntry: ParamList);
```

The first argument is the same as that used by application page programs. (The `kbint` and `net` options may not apply.) The `init` call occurs the first time that the program is invoked since being enabled. The `term` call occurs when the LA is being disabled. The `cycle` call is the normal one given each 15 Hz cycle or whenever directed via a special Data Access Table entry.

The second argument is a ptr to a part of the `LATBL` entry. It points to a structure in the table entry reserved for a ptr to the LA's static variables and an array of up to 10 integers, the first of which specifies the local binary Bit used as the enable/disable control for this invocation. The other integer array elements may be anything else required by the particular LA.

During the `init` call, the LA is expected to allocate memory for its own static variable requirements. This can be done by calling this routine:

```
Function Alloc(sVarSize: Longint): sVarPtrType;
```

This call invokes the pSOS memory allocation routine and returns a ptr to the allocated memory. If the storage cannot be allocated, a NIL ptr is returned.

When the `sVarPtr` is returned by `Alloc`, the LA should save it in the first longword of its `ParamList` structure. This is necessary because the LA is a Pascal procedure that must be invoked multiple times during the time that its `LATBL` entry is enabled. Any information that must be saved by the LA across calls to it must be stored in this static variable space. Note that a given LA may be invoked multiple times with different ParamList structures. An example is the Linac rf gradient regulation that is to be done for 3 rf stations by one local station. This will use 3 entries in `LATBL` but only a single entry in `CODES` for the gradient regulation program (procedure).

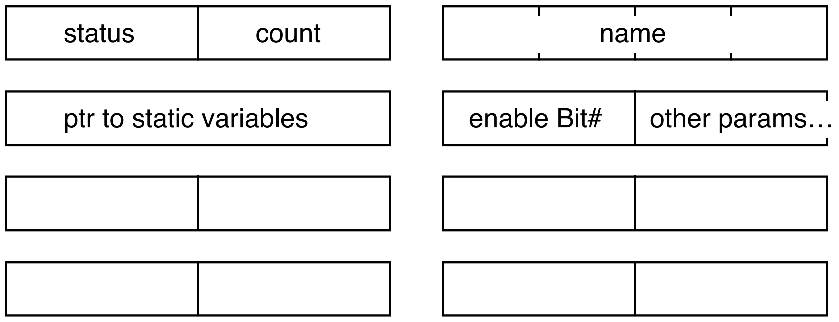
When the `term` call is made, the LA should free its static variable allocation by calling this routine:

```
   Procedure Free(statVarPtr);
```

This procedure simply frees the memory allocated by `Alloc`.

Local Applications Table

A new system table (#14) supports local applications. An entry in this table has the following format:



The status word is a copy of the previous enable bit reading. Comparing this value with the current enable bit reading allows the system logic to decide what to use for the `trig` argument in the call to the LA. The enable bit, when set, signifies that the entry in `LATBL` is enabled. When an LA entry makes a transition from disabled to enabled, the `init` call is used. The LA is expected to allocate any required static variables during this first execution. As long as it continues to remain enabled, the `cycle` call is used. When it makes a transition from enabled to disabled, the `term` call is used. Since the LA frees any static variable space during this execution, the act of disabling a local application means it will “start over” when it is re-enabled.

The program to be run is identified by 4-character name. Along with the type code of `LOOP`, the `CODES` table of downloaded separately-compiled programs is searched for a match, and the address of the executable copy of the program is used as the target for the call. The first time that the program is accessed, for the `init` call, a checksum check is performed to insure that the downloaded code has not been corrupted, and the program is copied into newly-allocated dynamic memory for execution. This means that the downloadable area is always available to receive a new version.

Downloading a new LA

When a change is to be made in a LA program, the new code is downloaded without concern for the currently-executing code. The LA scan finds the name of the LA and the ptr to the executing code (in on-board memory) in the `CODES` table entry corresponding to that name. The process of downloading leaves this pointer alone while the code is copied into a newly-allocated area.

When downloading is complete, the checksum is sent to be stored in the `CODES` table entry, and the ptr to the downloaded code is marked (by setting its ls bit) to show that it is a fresh copy.

LATBL ***table processing***

When LATBL entries are scanned by the Update task, and a fresh downloaded copy of the code is detected, and the type of call was to be a cycle call, the call type is altered to a term call. This gives a chance for the LA to free any allocated static variable storage and "clean up its act" in general. After any term call, the saved copy of the LA's enable bit reading is cleared. This will cause an init call to be given on the next cycle if the LA is still enabled. The checksum will again be checked and new memory allocated for execution in on-board ram.

After all LATBL entries have been scanned, a separate scan is made over the LOOP entries in the CODES table. For each entry which has the fresh download bit set, the bit is cleared, the executable area is freed and its ptr cleared.

The result of the above logic is that those entries which use the program just downloaded will be disabled and re-enabled automatically the very next cycle. (If it is desired to prevent an alarm message from being sent, in the case that the enable/disable status bit is being monitored, one can merely elect to use the 2X option with that status bit, since the bit will be disabled for only one cycle.) This means the new version of the code will take effect right away. To prevent this, either disable all LATBL processing by disabling the Data Access Table entry, or disable all LA entries which use the program to be downloaded. The local application may prevent this by disabling itself during term processing.

PAGEP ***table processing***

The index page logic directs the call-up of application pages. When a page is being called up, if the lo byte of the longword which contains the pointer to the entry point of the application page program is nonzero, the 4-byte "pointer" is assumed to be a program name of type PAGE. (Note that this implies that using a ptr in the old way can still work as long as the entry point address is on a 256-byte boundary.) A search is made for a match in the CODES table, the download area is sum-checked, memory is allocated in on-board ram for it, and the program is copied to that area for execution.

When the program terminates, either because a new page is called up or a return is made to the index page, a scan is made of the CODES table. The allocated area of any PAGE type entry in the CODES table is freed, and its pointer is cleared. This is done because only one PAGE program can run at a time. Note that this is in contrast to the LOOP type programs, in which many can be running at once.

TASK ***or*** ***INIT*** ***processing***

New tasks may be added to the system code by making a scan at reset time which looks through the CODES table for any entries of type TASK or INIT. Such entries can be copied into executable memory and called. What they do depends upon how they are written. Such a program could spawn and activate a task, for example. Or it could simply do some job at reset time. Only one call would be made to such a program, and it would be made from the ROOT task. This adds another dimension to system configuration possibilities.